Reactive Objects

Frédéric Boussinot, Guillaume Doumenc, Jean-Bernard Stefani

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Reactive Objects

Frédéric Boussinot , Guillaume Doumenc , Jean-Bernard Stefani

N° 2664

Octobre 1995

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

Reactive Objects

Frédéric Boussinot , Guillaume Doumenc , Jean-Bernard Stefani

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet MEIJE

Rapport de recherche n ° 2664 — Octobre 1995 — 23 pages

Abstract: In the reactive approach, system components are not supposed to execute at their own rate, but are instead driven by a logical common clock, defining global instants. The Reactive Object Model introduced in this paper, is an object based formalism matching the reactive paradigm. In this model, methods can be invoked using instantaneous non-blocking send orders, which are immediately processed (that is, processed during the current instant); moreover, a method cannot execute more than once at each instant. The Reactive Object Model is described and compared to the Actor Model; then a prototype language based on this model is introduced; finally its expressive power is shown on the example of a broadcast communication mechanism.

Key-words: Actors, Concurrent Programming,Objects, Reactive Approach

(Résumé : tsvp)

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Objets réactifs

Résumé : L'approche réactive suppose que les composants d'un système parallèles ne s'executent pas à leur propre rythme, mais sont dirigés par une horloge logique qui définit des instants globaux. Le modèle des Objets réactifs introduit dans ce papier est un formalisme "basé-objets" fondé sur le paradigme réactif. Dans ce modèle, les méthodes sont appelées par des odres d'exécution instantanés et non bloquants, traités immeédiatement (durant l'instant courant). De plus, une même méthode ne peut s'exécuter plus d'une fois durant le même instant. Le modèle des objets réactifs est décrit et comparé au modèle des Acteurs, puis un langage prototype fondeé sur ce modèle est introduit. Pour terminer, on montre la puissance d'expression du modèle à travers l'exemple d'une communication diffusée.

Mots-clé : Acteurs, Programmation concurrente, Objets, Approche réactive

1 Introduction

In usual programming languages (for example, C), one of the main reasons for defining procedures is to allow *reuse* of pieces of code without rewriting them. After being defined, a procedure has just to be called instead of copying its body. Moreover procedures are the basic units for modularity (even if they are generally felt as not sufficient for that purpose). Usual languages provide only *sequential* programming constructs: there is only one control flow, which in case of procedure calls, is suspended in the caller until the called procedure terminates; this "Procedure Call" paradigm is shown on figure 1.

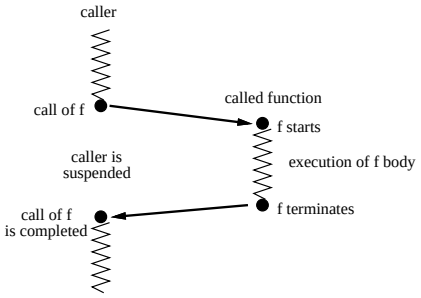

Figure 1: The Procedure Call Paradigm

Notice that the "Remote Procedure Call" (RPC) protocol[7] is a natural extension of the Procedure Call paradigm to distributed computing. In this context, the client is the caller which calls the server. The RPC paradigm is shown on figure 2.

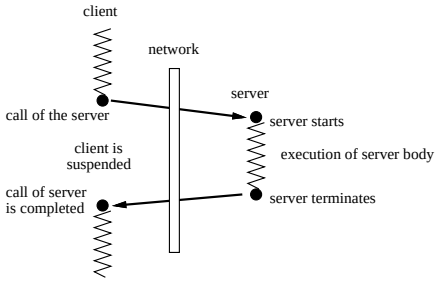

Figure 2: The Remote Procedure Call Paradigm

Many extensions of the Procedure Call paradigm have been designed to allow several control flows. One of the first extension is the *Actor* model originally proposed by Hewitt[11], and later developed by Agha[1]. Actors are parallel autonomous agents, which are distributed in space and execute at their own rate (thus, each actor represents a distinct control flow) and communicate asynchronously by sending messages. The send primitive is the non-blocking analog of procedure call: in the Actor model, a caller just *send* an execution order (a message) to a called actor, and it continues without waiting for the called actor to receive (or process) the order. Thus, the caller and the called actors are logically executed in parallel, as soon as the execution order is sent by the caller. This "Send and Forget" paradigm, which is the core of the Actor model, is shown on figure 3.

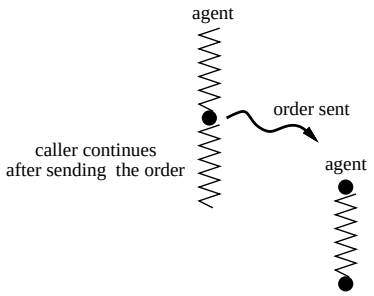

Figure 3: The Send and Forget Paradigm

Several questions arise from this paradigm:

1. After being sent, when will an order actually be executed ?
2. How to get a result from the called agent ?
3. How to deal with *concurrent* orders sent to the same agent ?

We are now going to consider these questions in turn.

For being executed, an order first has to reach the called agent; thus, here is a need to insure that an order will *eventually be delivered* (else programming would become quite problematic). In the Actor model, messages are buffered into *mailboxes*, from which actors take their inputs, and a *fairness* assumption states that every message sent to an actor is guaranteed to be eventually put into its mailbox[2]. An important point is that there is no way for the caller to get information on what really happens, and when it happens. Notice that in a distributed context, fairness simply means that the communication network is not allowed to loose messages.

To return a result from a called agent to the caller, is not as simple as in the Procedure Call paradigm, since there is no implicit synchronization between them at the end of called agent execution. However, the benefit is that the caller blocks waiting for the result, only when it really needs it. A standard solution is to send along with the order an extra information used by the called agent to return the result to the caller. It looks like figure 4, in which the caller transmits its identity to

the called agent to allow it to reply (this means that not only "simple" values are to be associated to orders, but also "higher order" ones, like *agent identities*; this has deep implications for semantics; see[14] for a full discussion on this aspect).

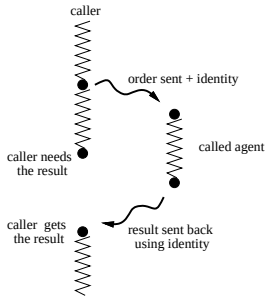

Figure 4: Getting Results from Agents

As parallelism is basic in the Send-and-Forget paradigm, there must be a way to deal with concurrent orders sent to the same agent. Figure 5 shows a situation where two distinct agents send two orders to the same target. In the Actor model, the problem is solved by the use of mailboxes which buffer messages.

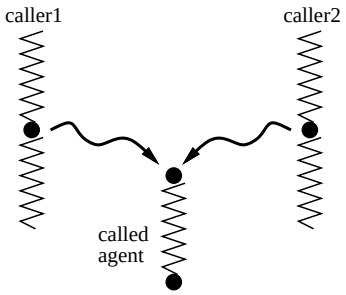

Figure 5: Several Calls to a Same Agent

Several problems are raised by the Actor model:

- A fairness hypothesis is actually not sufficient as it does not implies that transmission delays (between the sending of a message and its placing into a mailbox) are *bounded* (it only implies that they are finite). In real situations, something that can be delayed during an unbounded amount of time is often considered as equivalent to one allowed to never occur. In short, fairness means finite but unbounded, but what is really sought is a kind of bounded delay property.
- Program behaviors may depend on mailbox sizes. Moreover, overflow problems arise when using bounded size files; on the other hand, using unbounded files may lead to situations where unbounded memory allocation becomes needed.

- The need for mechanisms to overpass the mailbox mechanism may appear in some situations, for example to deal with priorities (alarms, for instance). The *express* mode of ABCL/1[16] is an example of such a mechanism.

We are now going to introduce the notion of an instant into the Send-and-Forget paradigm. to obtain a new paradigm called "Reactive Agent" paradigm.

2 The Reactive Approach

In the reactive approach, processes (as we call parallel system components) are not allowed to execute at their own rate, but are driven by a logical "basic clock" which defines a common time. In other words, there are *instants* which are *global* to all processes, and a particular process is not allowed to execute for the next instant while there exists another process which is not completed for the current instant.

Several models are based on the reactive approach, which mainly differ on the way processes are created and communicate:

- In *Nets of Reactive Processes* (NRP)[6], processes are connected through channels, to form deterministic nets. Channels are infinite "first in/first out" files; a process can output messages into channels (always possible, as channels are infinite) or it can input messages from channels, being blocked when channels are empty. Each channel has at most one producer process and at most one consumer process, and nets can be recursively defined (thus, channels and processes can be dynamically created). The key point of the NRP model is that processes are also allowed to test for *channel emptyness during one instant*, while preserving net determinism.

- In the POR/RLib formalism, processes are dynamically created and triggered by broadcast events. The POR/RLib is the kernel of an industrial process control tool (used to pilot Lafarge cement factories)[8].

- In the *synchronous language SL*[9], agents communicate using broadcast signals[1]. At each instant, a signal is either present, that is emitted by some agent during this instant, or absent otherwise; moreover, all agents get the same information about the signal presence or absence (this is the broadcast characteristic). There is no dynamic creation of agents in SL.

The reactive approach focus on the notion of logical time, common to several processes. It naturally leads to decompose complex systems into several "*reactive areas*", each of them defining its proper common time. Reactive formalisms are well suited to program these reactive areas, while standard asynchronous mechanisms (like for example, message passing) are reserved for inter-areas communications.

[1] SL is strongly linked to the Esterel synchronous language[4].

The Reactive Agent Paradigm

Introducing instants into the Send-and-Forget paradigm, allows to use them to bound delays of order transmissions. More precisely, transmission of an order can be required to be processed *during the same instant* the order is sent. These one-instant transmission orders will be simply called *instantaneous orders.*

To preserve the globality of instants, it is natural to require that an agent does not process more than one order during one instant (otherwise, one agent could execute several of its instants while others could execute none). This solves the concurrency problem raised by concurrent calls to the same agent: at each instant, only one order may be processed by an agent, the other ones being simply rejected. Let us call this, "*one call per instant*" property. An important point is that a caller agent can immediately (that is, in the current instant) know if its orders are rejected or accepted.

Instants allow to decompose agent behaviors into steps. During a given global instant, an agent may be called or not; in the first case, it executes its next step, and otherwise, it remains asleep. A pathological situation is when an agent execution step does not terminate; in this case, the agent will never be able to react to any new execution order; notice that in such situations, globality of instants prevents *the whole system* to terminate for the current instant, and thus, to go to the next instant. We will say that an agent has the "*reactivity*" property if all its execution steps always terminate, and is thus ready to react to orders at each instant.

Thus, the Reactive Agent paradigm consists of agents reacting to instantaneous orders and verifying the "one call per instant" and the "reactivity" properties. This leads to an approach in which systems are decomposed into reactive areas where agents communicate inside their area by instantaneous orders, and between distinct areas, by usual unbounded delay orders. This is shown on figure 6.

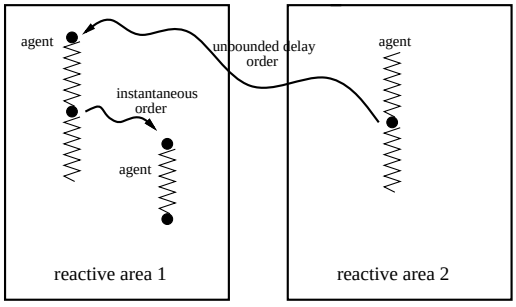

Figure 6: The Reactive Agent Based Approach

Notice that mailboxes are useless for reactive agents, as orders need not be stored: at each instant, if an order is the first one addressed to a target agent, then it can immediately fire it; otherwise it is simply rejected. Notice also that the fairness assumption of the Actor model is no more needed inside reactive areas, where instantaneous orders are used.

An important problem in parallel programming is to avoid inter-blocking between mutual calls. We end this section, with an example showing the use of the "one call per instant" property to avoid inter-blocking situations. We consider two graphical objects `O1` and `O2` shown on figure 7, that are linked, and must move together. Each object reacts to a `move` order and immediately transmits it to the other object. Cycles[2] between `move` calls are automatically broken because of the "one call per instant" method property; for example, if `O1` receive a `move` order, it transmits it to `O2` which in turn, transmit the order to `O1`; but the transmission cycle stops then, as `O1 move` method has already been executed for the current instant.

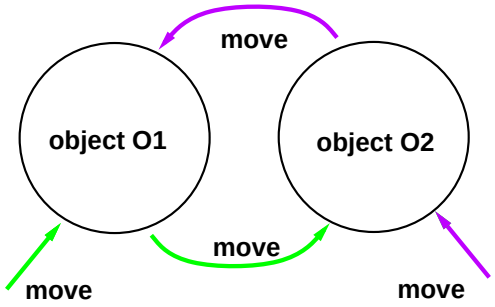

Figure 7: Two Linked Graphical Objects

3 Reactive Objects

We are now defining a model based on the Reactive Agent paradigm, and called "*Reactive Object Model*" (ROM). In this model, objects encapsulate data shared by agents, called methods, owned by the object. Methods can be invoked by instantaneous orders, have the "one call per instant" property, and are intented to be reactive. This model is currently studied and developed under contract with the France Télécom company[3]. Its formal semantics is described in [14].

First, we introduce the model; then section 4 describes a prototype language based on it; in section 5, we use the language to encode a broadcast communication mechanism; finally, future works are considered in section ??.

Objects, Methods and Systems

An *object* encapsulates data shared by all its *methods*, which are parallel and concurrent treatments on these data. A *system* defines a reactive area, and is made of objects that are run in parallel and share the same temporal reference. There are thus three levels: systems, objects in systems, and methods associated to objects; the key point is that the same notions of an *instant* and of a *phase* go through the three levels:

[2] called "causality cycles" in synchronous languages.
[3] Contract France Télécom-CNET 93 1B 141, #506

- Execution of a method is divided into instants: one can speak of the first instant of the method, of the second instant, and so on. The same method cannot be run several times during one instant. Moreover, a method can *suspend* its execution, waiting for a new *phase*, and execution will resume in the current instant, when this phase will be reached (thus, phases could as well be named "*micro instants*").

- Execution of an object for the current instant is terminated when all its methods have finished to execute for that instant.

- A new instant takes place when all objects have terminated their execution for the current instant; a new phase takes place when all methods either have finished to execute for the current instant or are suspended waiting for a next phase.

Therefore, instants and phases provide systems with a *global synchronizing mechanism*: the system, its objects, and their methods all run to the same global clock, and no component is free to take some advance on the others.

Figure 8 shows the structure of an object, made of three parts:

- a data part defining object data;

- an interface part, for calling object methods; one can see it as made of "*push buttons*" : to call a method, one pushes the corresponding button, which remains depressed until next instant; moreover, all buttons are automatically released at the beginning of each new instant;

- a code part, made of several pieces of code, each one defining the body of a method. The arrow depicted with each method body shows where execution is currently stopped in it.

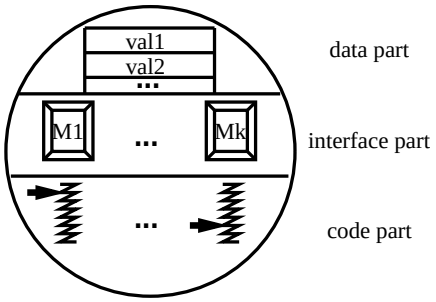

Figure 8: An Object

Method Calls

There exists three kinds of method calls;

- In *asynchronous* calls, the caller does not wait for the called method to terminate; it just sends an order and continues to execute.
- In *synchronous* calls, the caller halts until either the order is rejected, or, if accepted, the called method terminates for the current instant.
- *Delayed* calls are asynchronous calls which are delayed to the next instant.

In the two first cases, the caller can be informed whether the order sent is accepted or rejected (this information is meaningless for delayed calls which are processed only at next instant). Parameters may be transmitted during calls and they may be of any kind; they can even be objects or methods.

Figure 9 shows the effect of an accepted call of a method named `M` in object `O`: button `M` is pushed and execution of the method body associated to `M` goes to a new point shown by the arrow. On the other hand, figure 10 shows a rejected call to an already done method (the corresponding button is depressed): it has no effect at all (recall that all buttons are raised at the beginning of each new instant).

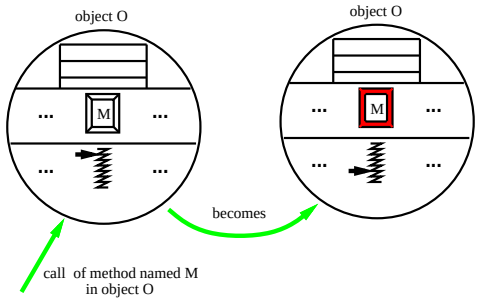

Figure 9: Accepted Send to a Method

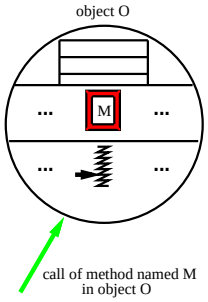

Figure 10: Rejected Send to a Method

Clones

Objects can be dynamically created as *clones* of existing ones[4]. A clone of an object O is a new object whose methods are copies of those of O. The cloning operation

[4]The notion of a clone comes from the SELF language[18]

is shown on figure 11. Notice that in the clone object, data are not copied and are undefined (the $\perp$ symbol), and that method bodies are reset.

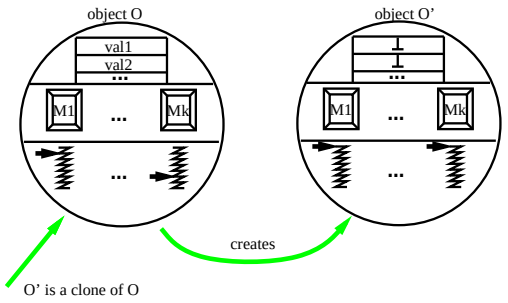

Figure 11: The Clone Operation

Adding Methods

Methods can be dynamically added to objects. The effect of adding a method to an object is shown on figure 12: a new button is added with a new reset instance of the method body (notice that button and method names are not necessary the same).

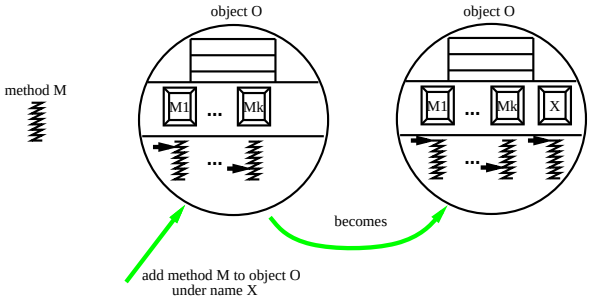

Figure 12: Add a Method

When adding a method, if there exists a method with the same name, then any access to the old method becomes impossible (thus, the old method becomes unaccessible).

Initial Object

There exists an *initial* objet, with only one method; this method is continuously executed by the system and each execution of it defines a new global instant. During one instant, objects are created and methods are called; execution of called methods can cause creation of new objects and calls of new methods, that will be executed in the same way. A new phase is issued when all methods either are suspended, or have finished to execute for the current instant. The current instant is terminated when all called methods have terminated their execution for that instant; then, the initial method is called another time, defining a new instant, and so on forever.

This concludes the Reactive Object Model description. The next section gives a syntax and defines a prototype language based on the ROM model.

4 The Language

This section gives a syntax for the ROM model. The language described is built on REACTIVE C (RC)[5] which is itself based on C, and gives a way to code systems designed according to the reactive approach. Our intention is **not** to give a full fledged language, but just to define a *prototype* that can be useful to investigate the ROM model. We are aware that this prototype language is not user-friendly, and we plan to build a real language based on the ROM model.

Object Definitions

The data part of an object is a structure whose fields can only be accessed by the object's methods. Definition of an object `O` having fields `f1,...,fk` is written:

```
object(O)
    f1;
    .....
    fk;
endobject(O)
```

Example :

```
object(Signal)
   int emitted;
endobject(Signal)
```

An object `O` is simply declared by:

```
objectType O;
```

In this case, `O` has to be defined as clone of another object (see bellow).

A field `x` can be accessed by object's methods under the name `field(x)`.

The object named `basic` has an empty data part, and is predefined.

Method Definitions

The definition of a method `M` associated to an object `O` has the following form:

```
method(O,M){
   ...
}
```

Method bodies are made of RC statements. A method is called `me` in its body, and its name is `me->name`. Variables defined in a method body are local to it and are introduced by the `var` keyword. The owner object of the method is called `self`, and its name is `self->name`. For example, the following method prints its name each time it is executed:

```
method(basic,identify){
   for(;;){
      printf("I am %s. ",me->name); }
      stop;
   }
}
```

Notice the `stop` RC statement that stops execution for the current instant; execution will restart from this point, when the method will execute for the next time.

Remark also that there is the possibility to define non-reactive objects whose method never terminate for the current instant, an example of which would be `identify` without the `stop` in it. Such objects are definitely erroneous in the Reactive Object Model.

Clones

The clone `New` of an object `Old` is created by the definition:

```
clone(Old,New);
```

Object `New` is local to the block in which the definition takes place.

An already declared object `New` is defined as a clone of `Old` by executing the statement:

```
isclone(Old,New);
```

Adding Methods

To add a method `M` to object `O` under name `X` (which is a string) is written:

```
addas(O,M,X);
```

When `X` is the name of `M`, one simply writes:

```
add(O,M);
```

Asynchronous Calls

Asynchronous method calls are performed using the `send` statement. A parameter is associated to an execution order; it is a pointer of type `void*`, and `NULL` denotes

a parameter which is not useful. If the call is accepted, the parameter is transmitted to the called method, and in the method body, it is known as `arg`. An asynchronous call to a method of object `O`, whose interface name is `"name"`, with parameter `arg`, is written:

```
send(O,"name",arg);
```

Example :

```
send(O1,"identify",NULL);
```

To get information on acceptance or rejection of an asynchronous call, one uses the `sendres` primitive. Code `UNKNOWN` means that the method is unknown, and `ALREADY_DONE` that the call is rejected because the method has already been called during the current instant. The return code is assigned to an `int` variable given as fourth argument; for example:

```
sendres(O,"name",arg,result);
```

Synchronous Calls

In a synchronous accepted call, the caller waits for the called method to terminates for the current instant. A synchronous call to a method of object `O`, whose interface name is `"name"`, with parameter `arg`, is written:

```
call(O,"name",arg);
```

To get information on acceptance or rejection of a synchronous call, one uses the `callres` primitive; for example:

```
callres(O,"name",arg,result);
```

Delayed Calls

A delayed call for the next instant is made using the `sendnext` primitive. There is no corresponding primitive to get information about acceptance or rejection of the call, as it does not concern the current instant. A delayed call to a method of object `O`, whose interface name is `"name"`, with parameter `arg`, is written:

```
sendnext(O,"name",arg);
```

Method Renaming

A method can dynamically change its interface name by executing a renaming statement. For example, it is changed in `"new"` by:

```
rename("new");
```

Renamings are specially useful to restrict access to methods. For that purpose, a method can rename itself by a new name which will be communicated only to objects allowed to call it. For example, consider the following method:

```
method(O,public){
   rename(arg);
   ...
}
```

One gets a restricted access to the method, by calling it with a new name as parameter (one supposes there is a way to create new fresh names); then, the method will not be accessible by the old name `public`, but only by those knowing the new name.

Initial Object

The *initial* object defines the system entry point; it owns a special method, also called `initial`, whose continuous execution defines the global instants. When it is needed, one can pass parameters to the system, using the standard "`argc/argv`" C mechanism. To define the initial method with the parameters `param` has the following syntax:

```
initial(param){
   ...
}
```

Example :

```
initial(int argc, char *argv[]){
   for(i = 0;;i++){
      printf("\ninstant %d: ",i);
      stop;
   }
}
```

Phases

Instants may be decomposed into phases. The notion of a phase is strongly related to the possibility in RC to decompose a reaction into several micro-reactions, using the `suspend` statement. We thus choose to directly use this RC statement, and get the two following primitives:

- The boolean test `isPhase(n)` is true if and only if the system in in phase number `n` (the initial phase has number 1).
- By executing the `suspend` statement, a method suspends its execution.

For example, one waits for the phase number 2 by executing the following loop:

```
for(;;){
   if (isPhase(2)) break;
   suspend;
}
```

A new phase is issued by the system when each called methods either is completely terminated, or has terminated its execution for the current instant (executing a `stop` statement), or is suspended (executing a `suspend` statement). A new instant is issued by the system when a new phase takes place while no suspended method remains. Moreover, the phase number is reset to 1 at the beginning of each new instant.

5 Expressivity

We are going to consider two examples to show the expressive power of the ROM model. In the first one, one defines some kind of "active" objects that can execute in an autonomous way. The second example shows how to implement a broadcast communication mechanism.

Active Objects

Using the `sendnext` statement, a method can continuously call itself for the next instant. An object owning such a method is an *active* object that executes in an autonomous way, without the need for another object to call one of its methods. For example, the following program defines an active object `O` which at each instant, prints a message, although its `run` method is only called once by the initial object:

```
method(basic,run){
   for(;;){
      printf("run!");
      sendnext(self,me->name,NULL);
      stop;
   }
}

initial(){
   clone(basic,O);
   add(O,run);
   send(O,"run",NULL);
}
```

Instantaneous Broadcast

We want to implement a very basic notion of a broadcast event, with only two primitives:

- `awaitEvent(E)` blocks control until event `E` is generated.
- `generateEvent(E)` generates event `E`, which has the effect to immediately resume execution of all methods waiting for `E`.

Notice that this is a real broadcast communication mechanism (the agent generating an event has no knowledge on the agents that are blocked waiting for it) and not a simple multicast (in which the event generator would have to know all the agents awaiting the event).

The implementation is based on the following points:

- Events are objects which have two associated methods `GenerateEvent` and `AwaitEvent`.
- The `GenerateEvent` method sets a field of the object generated.
- There are two phases: events are generated during the first one, and absences of events are processed during the second one.
- The `AwaitEvent` method tests continuously the object field; if the second phase is reached (the event is thus absent), then the method executes a delayed call to itself for continuing to work at next instant; otherwise, it terminates. Notice that events are examples of "active" objects.

Here is the code:

Events. An event is an object with only one field called `gen`.

```
object(BasicEvent)
  int gen;
endobject(BasicEvent)
```

To generate an event means to assign `gen` the number of the current instant, which is stored in the `_instantNumber` variable. This variable is automatically set by the system at the beginning of each global instant, and thus, presence of all events is automatically reset when a new instant takes place.

```
method(BasicEvent,GenerateEvent){
  for(;;){
    field(gen)= _instantNumber;
    stop;
  }
}
```

During phase 1, the `AwaitEvent` method continuously tests the event to be generated, and if it has not been generated when phase 2 takes place, it calls itself for the next instant. When `AwaitEvent` detects that the event is generated, it calls a method which has the same name than itself, and which belongs to an object passed as argument, and then it terminates. Code for `AwaitEvent` is:

```
method(BasicEvent,AwaitEvent){
  var objectType obj = (objectType)arg;
  for(;;){
    if(isPhase(2)){
      sendnext(self,me->name,NULL);
      stop;
    }else if(field(gen)!=_instantNumber){
      send(obj,me->name,NULL);
      return;
    }else suspend;
  }
}
```

Notice that the method continuously executes the `suspend` statement while phase 2 is not reached and the event is not generated.

Generation of an Event. Code corresponding to `generateEvent(evt)` is:

```
send(evt,"GenerateEvent",NULL);
```

Notice that to generate an already generated event has no effect as the `GenerateEvent` method is executed at most once during one instant ("one call per instant" property).

Waiting for an Event. A method waiting for an event `E`, begins by adding to `E` an instance of `AwaitEvent` under a new `secret` name; then, it calls it using a synchronous call; if it returns during phase 1, this means that the event is generated, and the `awaitEvent` statement terminates; if it returns during phase 2, the event is considered as definitely absent for the current instant; then the method renames itself in `secret` and executes a `stop` statement to terminate the current instant; in this case, only the event will possibly call it, when the event will be generated. Code corresponding to `awaitEvent(evt)` is[5]:

```
{
  var char *secret = newName(), *save;
  addas(evt,AwaitEvent,secret);
  call(evt,secret,self);
```

[5] The `newName` C function returns a new fresh name, and the `strdup` function copies character strings.

```
  if(isPhase(2)){
    save = (char*)strdup(me->name);
    rename(secret);
    stop;
    rename(save);
  }
}
```

Notice that the object identity (`self`) is the parameter of the synchronous call to the event, allowing it to reply to the object, when the event will be generated (this corresponds to figure 4). Notice also that in case method name is changed into `secret`, it is restored at the end of execution (after having been saved using the `strdup` function). Moreover, when the event is generated during first instant, call of the `secret` method by the event has no effect as this method does not exist in the object.

6 Reactive Objects and Distribution

The Reactive Object Model introduced above relies on a global notion of instant shared by all reactive objects. In a distributed context with potentially long, and even a priori unbounded communication delays, realizing a global instant may be costly. More precisely, the practicality of "instantaneous" reactions in the Reactive Object model, as with other synchronous formalisms such as ESTEREL, LUSTRE, etc., depends on the informal rule of thumb that reactions ought to be fast enough to capture all relevant event occurrences in the system. If a reaction involves a priori unbounded communication delays, then this rule of thumb will obviously be violated. Programming with reactive objects in distributed systems thus calls for some addition to the model.

As hinted at in Section 2, we can decompose a system into reactive areas, where agents communicate inside their area by instantaneous orders, and between distinct areas by usual unbounded delay orders. In this section, we discuss this model in more detail, and briefly comment on its implementation.

Reactive areas in unbounded delay environments

We consider first the basic model, with distinct reactive areas interconnected by a communication infrastructure with unbounded communication delays (the general case in distributed systems).

Communication between objects residing in two distinct synchronous areas is also by means of asynchronous and synchronous calls. In an asynchronous call, the caller does not wait for the called method to terminate, just as with the `send` primitive. Unlike an asynchronous call with the reactive `send` primitive, however, the execution of the invoked method does not take place within the same instant, but instead at an

arbitrary future one. For the same reason, a synchronous call between two distinct reactive aeras extends over an indeterminate number of instants.

From the point of view of the reactive model, synchronous calls to reactive objects outside the originating area terminate the execution of the calling method for the current instant. The caller will resume execution when the synchronous call returns, at a different, a priori unknown, future instant. Notice that it is possible in the reactive model to precisely measure, in terms of instants in the caller's reactive area, the duration of a synchronous call[6].

In this model, the semantics of calls is "timed", i.e. calls span several (at leat one) instant. This is what distinguishes calls within a reactive area from calls between different reactive areas. Although, semantically, the time considered is purely logical (instants derive from the operational semantics rules of the model and arise as terminated or stopped computations), it is possible to relate this logical time with physical time in order to maintain real-time synchronization. This can be achieved by anchoring the initial object in Section 4 to a time driven signal such as a real-time clock, and by ensuring that each reaction, i.e. each instant, in a reactive area is bounded in time. The time granularity of instants in a reactive area is thus essentially given by the bound on their execution time.

The informal rule of thumb referred to above can then be simply stated as "the smallest delay between two consecutive occurrences of the same event type should be greater than the time granularity of instants".

Reactive areas in bounded delay environments

The semantics of communication in unbounded-delay environments described above reflects the assumption that the programmer has no knowledge of, or does not rely on any knowledge of communication delays that may occur between reactive areas. In large distributed systems, this is in general the case, with the unbounded delay assumption abstracting assumptions about the implicit behavior of the communication infrastructure as well as potential failure modes (from the nodes in the system or from the communication infrastructure). In environments with a communication infrastructure providing bounded communication delays, however, it is possible to refine the basic distributed model by synchronizing different reactive areas.

In the synchronized model, reactive areas operate in parallel but their execution is synchronized so that their reactions occur during the same global instant. The semantics of calls between reactive areas is as presented above, except that they are now guaranteed to arrive at their destination at the next instant. For instance, when a synchronous call is made to a reactive object in a different reactive area, it triggers a reaction at the next instant following its initiation. Likewise, the response to a synchronous call will arrive at the caller in the initial area exactly two instants after the initiation of the call. Distributed executions in the system are thus globally

[6] For instance, it suffices to maintain some "clock variable". When the call returns, the clock variable holds the number of instants elapsed since the initiation of the call.

clocked, but communication and reaction are only instantaneous within a reactive area.

Communication between reactive area takes exactly one clock tick. This synchronized model represents one extreme, in which executions in the whole system is controlled by a global clock whose granularity is given by the sum of the bounds on communication delays and on reactive area reaction time. Other refinements are possible where, for instance, calls between different areas are delayed for some fixed, known number of clock ticks, allowing a finer granularity of time for the reaction of reactive areas.

Implementing Distributed Reactive Objects

The Distributed Reactive Object model (DROM) discussed above can be readily implemented using a standard distributed platform such as one consistent with the CORBA specification [15]. This entails deriving interface signature descriptions for reactive objects in the CORBA Interface Definition Language (IDL), and providing an appropriate binding between a reactive area and its environment.

Two related aspects must be considered in realizing the latter:

1. managing, i.e. generating and collecting, calls in and out of a reactive area,
2. preserving the semantics of reactions inside a reactive area.

Essentially, to comply with the semantics of calls explained above, the execution of each reaction inside a reactive area must be treated as a single atomic action, during which the environment visible from the area (i.e. set of incoming calls or responses from previous synchronous calls) must remain unchanged.

This is exactly similar to the construction, detailed in [17], of a virtual machine to execute programs written in the ESTEREL synchronous programming language. Indeed, a particular degenerated case of reactive area would be that of a single reactive object that could, e.g., be programmed using a standard synchronous language such as ESTEREL or LUSTRE.

7 Conclusion and Future Work

The Reactive Object Model is a new formalism that merges an object approach together with the notion of a global instant. Object methods are executed in parallel, and are called using instantaneous orders which are asynchronous calls processed at the same instant they are issued. A prototype language based on the ROM models have been designed and implemented in REACTIVE C. The expressive power of the ROM model is illustrated by implementation of a broadcast communication mechanism. Finally, the embedding of the ROM model into a distributed framework (DROM) has been discussed.

Work on the DROM model is currently under way. We plan to realize the DROM model as described in Section 6 by integrating the ROM model on top of a CORBA [15] and ODP [10, 3] conformant distributed platform (called TORB for Telecommunications Object Request Broker) [13], reusing work already carried out for the execution of reactive objects in a distributed environment based on the CHORUS operating system microkernel [17].

We also plan to integrate the DROM model with a study of *predictability* conditions, to be able to satisfy so called "quality of services" requirements (from a time related, as well as from a functional viewpoint)[12].

Finally, we plan to design a true syntax for the prototype language presented in section 4. This language, called RC++, would be built on the top of C++, and it would implement ROM reactive objects as well as broadcast communication.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha, S. Frolund, Woo Young Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. In Gul Agha and Peter Wegner and Akinori Yonezawa, editor, *Research Directions in Concurrent Object Oriented Programming*, pages 3–21. The MIT Press, 1993.

[3] ODP Reference Model: Architecture. Itu-t iso/iec recommendation x.903, international standard 10746-2, January 1995.

[4] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[5] F. Boussinot. Reactive C: An Extension of C to Program Reactive Systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[6] F. Boussinot. Réseaux de Processus Réactifs. Technical Report 1588, INRIA, 1992.

[7] J.R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Sun Technical Reference Library. Springer Verlag, 1991.

[8] G. Doumenc, G. Garcin, and P. Gueydon. Un Système de Contrôle/Commande Réactif Strict. In *Proc. RTS'94*, 1994.

[9] F. Boussinot and R. de Simone. The SL Synchronous Language. Technical Report RR-2510, INRIA, March 1995.

[10] ODP Reference Model: Foundations. Itu-t iso/iec recommendation x.902, international standard 10746-2, January 1995.

[11] C. Hewitt. Viewing Control Structure as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[12] J.B. Stefani. Computational Aspects of QoS in an Object Based Distributed System Architecture. *3rd International Workshop on Responsive Computer Systems, Lincoln, NH, USA*, September 1993.

[13] J.B. Stefani, and P. Auzimour, and F. Dang Tran, and L. Hazard, and F. Horn, and V. Perebaskine. A Real-Time DPE on top of the Chorus micro-kernel. Technical Report NT/PAA/TSA/TLR/4179, CNET, January 1995.

[14] C. Laneve and F. Boussinot. Two Semantics for a Language of Reactive Objects. Technical Report RR-2511, INRIA, March 1995.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification. *OMG Document 91.12.1*, December 1991.

[16] A. Onezawa, J.P. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *Proc. of the OOPSLA'86*, pages 258–268, 1986.

[17] R. Bernhard and L. Hazard and F. Horn and J.B. Stefani. Implementation of a Synchronous Execution Machine on the Chorus micro-kernel. In *Proceedings 14th IEEE Real-Time Systems Symposium, Raleigh, NC, USA*, December 1993.

[18] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. *Lisp and Symbolic Computation*, pages 187–205, 1991.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399